Partial Evaluation for Haskell

Extended Abstract

Duncan Coutts

Introduction This paper gives a brief introduction to partial evaluation of typed functional languages and presents initial research into building a practical partial evaluator for a first-order subset of Haskell using the Template Haskell infrastructure.

Partial evaluation has traditionally been done for untyped strict functional languages such as Scheme. We look at the difficulties in finding a feasible type for our partial evaluation function that avoids using too many layers of encoding and present a solution using generating extensions. We then look at constructing the generating extension from a suitably annotated original and the ease with which this can be implemented this using the Template Haskell infrastructure.

Partial evaluation Partial evaluation it a technique for producing faster programs by specialising them to some of their inputs (see [3] for an good introduction). From a program optimisation point of view partial evaluation can be seen as a very aggressive form of unfolding/unrolling, inlining and constant propagation. Another viewpoint is to note that a partial evaluator can convert a two-input program into a *staged* program that accepts one input and produces another program that accepts the other input and calculates the final answer; the point being that the first stage may be run ahead of time (e.g. at compile time) while the second specialised stage should run faster than the original program.

One of the very promising things about the technique is the ability to produce compilers from interpreters. A great deal of past research has gone into producing *optimal* partial evaluators that can eliminate all the interpretive overhead in the programs produced by such compilers.

The problem with types Research in partial evaluation often concentrates on using multiple self-application of the partial evaluator. This is a technique that becomes difficult in a typed setting and can easily lead to requiring double encodings of values that the partial evaluator manipulates. Indeed, previous investigations[1] of partial evaluation for Haskell have run into severe performance difficulties due to exactly this issue. With this in mind the first part of the paper considers the feasible types that a partial evaluator could have.

In this investigation we follow a suggestion due to Launchbury[4] to directly write the function “cogen” whereas by contrast, traditional techniques derive this function by triple self-application of the partial evaluator. This side-steps the issue of self-application and double encodings and it gives us a partial evaluation function with a feasible type. This “cogen” function does much the same job as a traditional partial evaluator but works more like a compiler, whereas a traditional partial evaluator is very much like an interpreter. `cogen` converts the input program into its *generating extension*, that is a two stage version of the original.

The difficulty in writing a compiler-style partial evaluator is not as great as one might think. In fact for Haskell it is easier than a traditional partial evaluator since is not directly related to the complexity of the language. A “compiler style” partial evaluator can reuse an existing compiler without modification, whereas a traditional “interpreter style” partial evaluator must embed an interpreter for the source language. This may not be a problem for a simple language from the LISP family but Haskell is significantly more complicated and writing an interpreter would be a significant project. It turns out that a simple “compiler style” partial evaluator for Haskell can be written in about 100 lines of interesting code (plus a further 300 of boilerplate code for traversing over abstract syntax trees).

Implementation using Template Haskell Part of the reason for the easy of implementation is due to our use of Template Haskell[2] which provides an abstract syntax tree, parser, and code generator. Most importantly the Template Haskell language allows one to write staged Haskell programs which is exactly what the programs output from `cogen` are. Furthermore because we are working in the context of an existing optimising compiler (GHC) we do not have to pay too much attention to the quality of the generated code since the compiler will do much of it for us, for example by inlining trivial functions.

The current implementation works for a range of small examples including producing a compiler from an interpreter for a simple imperative language. Another advantage of using Template Haskell is that it allows the partial evaluator to be implemented as a library rather than a source to source preprocessor phase. Here is an example of how we use the partial evaluator on an interpreter:

```
compiler = $(cogen [| interpreter |])
program  = $(compiler progAST)
```

The brackets are Template Haskell notation. `$( ... )` means "run the quoted function and paste in the generated code" while `[| ... |]` means "the abstract syntax of the quoted term".

Limitations and future work The most important limitation is that the programmer must fully annotate the program to specify which parts of the program are static and which parts are dynamic. To compound the difficulty of writing these annotated programs there is no check at the moment for the consistency of the annotations. But if the annotations are inconsistent then the generating extension will be ill typed. The solution to these two problems is binding type checking to check the consistency of the programmer-supplied annotations, and binding time inference to remove the need for many of the annotations.

The current implementation has limited support for dealing with partially static data. Useful examples include lists of static length but with dynamic elements. Applications for this extension include optimising vector and matrix calculations for vectors and matrices of statically known size. In summing vectors for example we can eliminate the intermediate lists that are created during normal evaluation.

A limitation of the current implementation technique is that the partial evaluator is limited to specialising programs that manipulate only first order values. This is because of the need to perform equality checks on the static values manipulated by the program. Currently, higher order values must be marked dynamic and so no specialisation improvements can be achieved for code that use them. In future work I hope to extend the technique to allow higher order data values by automaitcally converting code to use explicit closures which can be checked for equality.

The goal of this research is to apply this tool to existing embedded domain specific languages to try an achieve real performance improvements and to extend the tool as the applications demand.

References

[1] Silvano Dal-Zilio and John Hughes. A self-applicable partial evaluator for a subset of Haskell. August 1993. Available from http://www.cmi.univ-mrs.fr/~dalzilio/haskell-parteval.html

[2] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In ACM SIGPLAN Haskell Workshop 2002

[3] N.D. Jones, C.K. Gomard and P. Sestoft. Partial Evaluation and Automatic Program Generation. Prentice Hall International, 1993. Available from http://www.dina.dk/~sestoft/pebook/

[4] J. Launchbury. A strongly-typed self-applicable partial evaluator. In J. Hughes (ed.), Functional Programming Languages and Computer Architectures, August 1991.